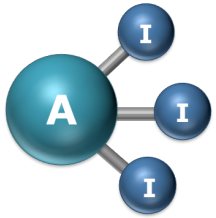


AI 4 Scientific Discovery Network+

PSDS
PHYSICAL SCIENCES DATA-SCIENCE SERVICE

```
IMPORT TKINTER

ROOT = TKINTER.TK()
ROOT.TITLE('ERRANTSCIENCE.COM')
SKETCHPAD = TKINTER.CANVAS(ROOT)

SKETCHPAD.CREATE_OVAL(100,50,150,100)

X,Y = 125,175
```


```
STICK = SKETCHPAD.CREATE_LINE(X,Y-75,X,Y)
ARMS = SKETCHPAD.CREATE_LINE(X-25,Y-50,X+25,Y-50)

DIFF_X = 25

LEGL = SKETCHPAD.CREATE_LINE(X,Y,X-DIFF,Y+50)
LEGR = SKETCHPAD.CREATE_LINE(X,Y,X+DIFF,Y+50)

SKETCHPAD.PACK()
```

Producing useful code

(Documentation, Typing, and Code Style in Python 3.5+)

Samuel Munday

This series has been co-created by the AI 4 Scientific Discovery Network & The Physical Sciences Data-Science Service

Reasoning about Code

- What is the author's *intent* behind this [line/function/library]?
  - Having an answer to this question makes the code easier to use, easier to modify, and eas extend

- As scientists, our job is to 'do science'
- Our software is a tool to help us, but not what we are there for
- This leads to 'quick and dirty' code
  - Difficult to go back and reason about our own code, let alone somebody else's!

“The most important single aspect of software development is to be clear about what you are trying to build”

Bjarne Stroustrup (Inventor of C++)

- If I don’t understand what your code does, I’m not going to use it
- If you don’t understand what your code does, you aren’t going to use it
- *Time you spend doing it now will be vastly outweighed by the time you (and others) save in future!*

How do we help people reason about our code?

1. ~~Write better code~~
2. Write good documentation
3. Make use of our Type System
4. Write with appropriate "style"

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Example: Lennard-Jones Simulation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def time2reduced(t):
    return t * np.sqrt(eps/(m*sigma**2))
def reduced2t(red):
    return red / np.sqrt(eps/(m*sigma**2))
def force2reduced(f):
    f * sigma / eps

n_atoms = 24
n_atoms_group = 4
n_groups = n_atoms / n_atoms_group
n_steps = 100
dt = 1E-13
t0 = 0
eps = 1.65E-21
sigma = 3.4E-10
m = 6.63E-26
boxsize= 5

locations = np.zeros((n_atoms, 3))
for group in range(int(n_groups)):
    locations[group*n_atoms_group:(group+1)*n_atoms_group, 0] = np.arange(n_atoms_group) - np.mean(np.arange(n_atoms_group))
    locations[group*n_atoms_group:(group+1)*n_atoms_group, 1] = group - n_groups/2
    locations[group*n_atoms_group:(group+1)*n_atoms_group, 2] = group % 2

locations *= 2
all_locations = np.zeros((n_steps, n_atoms, 5))
velocities = np.zeros_like(locations)
velocities += np.random.normal(0, 0.01, velocities.shape)
accelerations = np.zeros_like(locations)
forces = np.zeros((n_atoms, n_atoms, 3))
times = np.zeros(n_steps)
```

Better documentation would help us answer these questions!

Example: Lennard-Jones Simulation

```
def getforce(r1, r2):
    rsq = np.linalg.norm(r1-r2)
    return 4*((1/rsq**6) - (1/rsq**3))

def getaccelerations(forces):
    return forces.sum(axis=1)

def getforces(locations):
    for i in range(n_atoms):
        for j in range(i+1, n_atoms):
            r1 = locations[i]
            r2 = locations[j]
            displacement = r1-r2
            forces[i,j] = displacement * getforce(r1, r2)
            forces[j,i] = - forces[i,j]
    return forces

def do_timestep(t, locations, velocities, forces, accelerations):
    times[step] = t
    locations += (velocities * dt + (1/2) * accelerations * dt**2)
    forces = getforces(locations)
    new_accelerations = getaccelerations(forces)
    velocities += (1/2)*(accelerations + new_accelerations) * dt
    accelerations = new_accelerations
    for i in range(n_atoms):
        if (np.abs(locations[i,:]) > boxsize).any():
            velocities[i] = -velocities[i]
    return locations, velocities, forces, accelerations

t = t0

for step in range(n_steps):
    all_locations[step, :, 0] = np.arange(n_atoms)
    all_locations[step, :, 1:4] = locations
    all_locations[step, :, 4] = step
    locations, velocities, forces, accelerations = do_timestep(t, locations, velocities, forces, accelerations)
    t += dt
```

Better documentation would help us answer these questions!

Documentation in Python

Inline Comments

- *What does this code block do?*
- Uses # to signify comment start

```
def do_timestep(t, locations, velocities, forces, accelerations):
    # Velocity verlet: https://en.wikipedia.org/wiki/Verlet_integration#Velocity_Verlet

    times[step] = t
    # Update location
    locations += (velocities * dt + (1/2) * accelerations * dt**2)

    # Get new accelerations
    forces = getforces(locations)
    new_accelerations = getaccelerations(forces)

    # Get new velocities
    velocities += (1/2)*(accelerations + new_accelerations) * dt

    accelerations = new_accelerations

    # If an atom has moved too far, pretend it has hit a wall and bounced elastically
    for i in range(n_atoms):
        if (np.abs(locations[i,:]) > boxsize).any():
            velocities[i] = -velocities[i] # Elastic collision

    return locations, velocities, forces, accelerations
```

Documentation in Python

Docstrings

- *What does this function do?*
- *Held within 3 sets of quotation marks*

```
def getforce(r1, r2):
    """
    Parameters
    ----------
    r1 : Location 1 (3d vector)
    r2 : Location 2 (3d vector)

    Returns
    -------
    magnitude : Size of force


    Given two locations r1 and r2, calculate the LJ force between them.
    LJ potential is given by:
        4E [(sigma/r)**12 - (sigma/r)**6]
    Where r is the magnitude of the displacement vector from r1 to r2

    We set sigma=r=1 for convenience

    We can get r^2 easily - taking square roots is slow so we don't need to
    """
    rsq = np.linalg.norm(r1-r2)
    magnitude = 4*((1/rsq**6) - (1/rsq**3))
    return magnitude
```

Accessing Docstrings

Shift + Tab

```
help(np.linspace)
```

```
Help on function linspace in module numpy:

linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
    Return evenly spaced numbers over a specified interval.

    Returns `num` evenly spaced samples, calculated over the
    interval [`start`, `stop`].

    The endpoint of the interval can optionally be excluded.

    .. versionchanged:: 1.16.0
        Non-scalar `start` and `stop` are now supported.

    Parameters
    ----------
    start : array_like
        The starting value of the sequence.
    stop : array_like
```


Skills 4 Scientists

Typing in Python

- Python is strongly[1], dynamically[2] typed
  [1] I can only do operations that are explicitly defined on that type
  [2] I can change a variable to a different type whenever I want
- We can still use *type hints* to make our intent clear
  - And use a *type checker* to enforce them
- This makes our code *type safe*

- It is still our job to write code that uses the types correctly. Just because we have added type hints to a function does not guarantee that it will handle those types correctly.

```
def add(a,b):
    return a + b
for i, j in [[1, 2], ['Hello', ' World'], [1, 'Hello']]:
    print(add(i,j))
```

```
3
Hello World

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-200-9c9b5b8b8979> in <module>
      2     return a + b
      3 for i, j in [[1, 2], ['Hello', ' World'], [1, 'Hello']]:
----> 4     print(add(i,j))

<ipython-input-200-9c9b5b8b8979> in add(a, b)
      1 def add(a,b):
----> 2     return a + b
      3 for i, j in [[1, 2], ['Hello', ' World'], [1, 'Hello']]:
      4     print(add(i,j))

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

A Note on Code Style

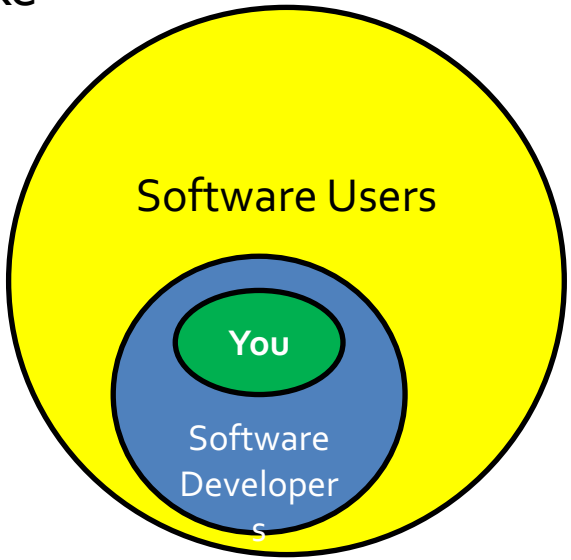
- We want our code to *look* like other people's
  - Makes it easier for users to read our code
- Python's style guide is described in ***PEP8***
  - Be aware that it exists
  - Use it as a reference
  - **Do not** drive yourself crazy adhering to it

- Two word summary: *Readability Counts*

Summary

- As scientific software developers, we have a duty to make sure our code is as *user-friendly* as possible
  - Make sure our intent is clear
  - Make sure our interfaces are clear

- The person who will spend the most time reading your code is **you**
  - 5 minutes now will save you hours in future



Resources

Books

- The Pragmatic Programmer (Hunt, Thomas)
- Python for Data Analysis (McKinney)
- Data Science from Scratch (Grus)
- The Hitchhiker's Guide to Python (Reitz, Schlusser)

Software Tools

- Mypy (Python Type Checker)
- Flake8 (Python Style Linter)

Skills4Scientists!